Simulating the Behaviour of the Human Brain on NVIDIA GPU:

cuHinesBatch & cuThomasBatch implementations

Author:
Ivan Martínez Pérez

Master in Innovation and Research in Informatics
High Performance Computing specialization

15 de Enero de 2018

Director:
Pedro Valero-Lara
Co-Director:
Antonio Peña

Facultad de Informática de Barcelona
Universidad Politécnica de Cataluña (UPC) - BarcelonaTech

Resumen

Abstract

Being able to understand the behavior of the Human Brain is one of the most important challenge of this decade. In this work we are going to present a small step towards this objective.

This work presents a novel advances in order to compute more efficiently the Hines algorithm on GPU, being this algorithm one of the most time consuming step in brain simulation. Moreover, all the work done bring us the opportunity to develop a more general tridiagonal solver, based on Thomas algorithm, which is presented in this work too.

This work is structured as follows. Section 2 briefly introduces what a GPU is and the different methodologies to deal with the problem at hand, that is the simulation of the behavior of the Human Brain. In Section 3 we present the advances proposed for the resolution of Thomas and Hines algorithms . Section 4 shows the performance achieved and finally the conclusions are outlined in Section 5.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Human Brain Project

The brain, with its billions of interconnected neurons, is without any doubt the most complex organ in the body. With 500 scientists at more than 100 universities, teaching hospitals, research centers across Europe and a total budged of 1 billion euros, The Human Brain Project, one of the Flagship European projects and the main frame where this Master Thesis is developed, proposes a completely new approach.

The project integrates everything we know about the brain into computer models and using these models to simulate the actual working of the brain. Ultimately, it will attempt to simulate the complete human brain. The models built by the project will cover all the different levels of brain organisation – from individual neurons through to the complete cortex. The goal is to bring about a revolution in neuroscience and medicine and to derive new information technologies directly from the architecture of the brain.

The challenges facing the project are huge. Neuroscience alone produces more than 60' 000 scientific papers every year. From this enormous mass of information, the project will have to select and harmonise the data it is going to use – ensuring that data produced with different methods is fully comparable. The data feeding the project's simulation effort will come from the clinic and from neuroscience experiments. As we try to fit all the information together, we will discover many of the brain's fundamental design secrets: the geometry and electrical behaviour of different classes of neurons, the way they connect to form circuits, and the way new functions emerge as more and more neurons connect. It is these principles, translated into mathematics that will drive the project's models and simulations.

Today, simulating a single neuron requires the full power of a laptop computer. But the brain has 100 billions of neurons and simulating all them simultaneously is a huge challenge. To get round this problem, the project will develop novel techniques of multi-level simulation in which only groups of neurons that are highly active are simulated in detail. But even in this way, simulating the complete human brain will require a computer a thousand times more powerful than the most powerful machine available today. That is the reason why Subproject 7 (High-Performance Analytics and Computing Platform), formed by specialists in supercomputing, is one of the key parts in the Human Brain Project. Their main task is to work with industry to provide the

project with the computing power it will need at each stage of its work.

The Human Brain Project will impact many different areas of society. Brain simulation will provide new insights into the basic causes of neurological diseases such as autism, depression, Parkinson's, and Alzheimer's. It will give us new ways of testing drugs and understanding the way they work. It will provide a test platform for new drugs that directly target the causes of disease and that have fewer side effects than current treatments. It will allow us to design prosthetic devices to help people with disabilities. The benefits are potentially huge. As world populations grow older, more than a third will be affected by some kind of brain disease. Brain simulation provides us with a powerful new strategy to tackle this problem.The project also promises to become a source of new Information Technologies. Unlike the computers of today, the brain has the ability to repair itself, to take decisions, to learn, and to think creatively - all while consuming no more energy than an electric light bulb. The Human Brain Project will bring these capabilities to a new generation of neuromorphic computing devices, with circuitry directly derived from the circuitry of the brain.

1.1.1 Arbor Simulator

Current simulators were designed for single core systems, with parallel implementations added later. There are efforts to add many core support to existing codes, however they are subject to the law of diminishing returns, this means that adding more cores could even cause a fall in performance. A common example is adding more people to a job, such as the assembly of a car on a factory floor. At some point, adding more workers causes problems such as workers getting in each other's way or frequently finding themselves waiting for access to a part.

This presents an opportunity to start work on the next generation of simulators, designed from the ground up to support diverse many core architectures. Led by ETH Zürich, Arbor[1], as one of the simulators born inside the philosophy of the Human Brain project, aims to fill this gap.

The simulation itself is divided into two big tasks, communication and computation, exchange and update-cells respectively on Figure 1.1. On communication, each simulated neuron sends the spikes generated on one simulation time step to the other interconnected neurons, the way that we determined if a spike is generated or not is through the computation phase. As we can see, the communication phase depends on the results obtained by the computation phase, leading us to one of the key points of Arbor implementation: each time step of the simulation is half step of the behavior that we are simulating, allowing a temporal pipeline implementation (Figure 1.2).

Despite being a great optimization that enables a increment in the parallelism, it is not the main focus of our thesis, that is the reason why we are going to shift our attention into the computation phase, the one that is in charge of determining the Voltage on neuron morphology and one of the most time consuming steps of the simulation.

[1] `https://github.com/eth-cscs/arbor`

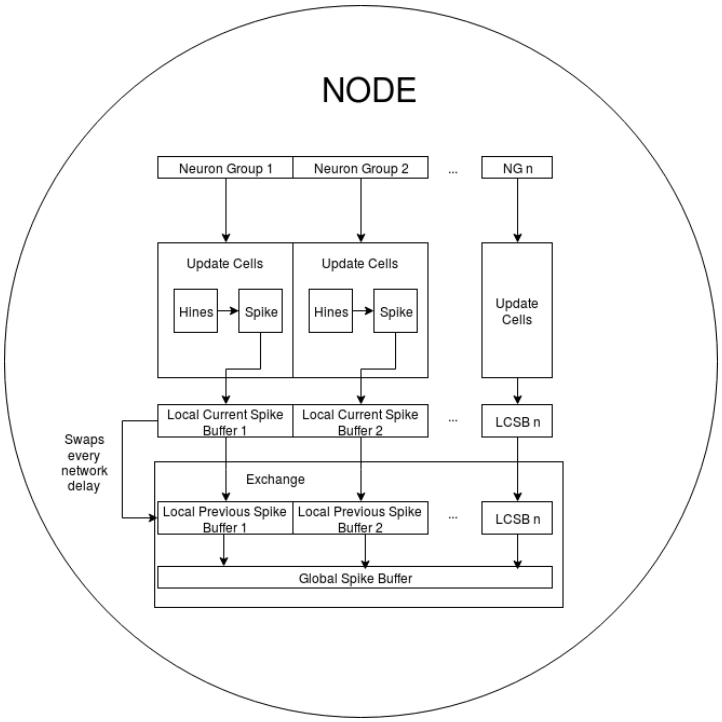

Figure 1.1: Arbor main behavior

	dt=0	dt=1	dt=2	dt=3
i=0	update_cells	exchange		
i=1		update_cells	exchange	
i=2			update_cells	exchange
i=3				update_cells

Figure 1.2: Arbor simulation pipeline

1.1.2 Hines Method

The standard algorithm used to compute the Voltage on neurons' morphology is the Hines algorithm [7]. This algorithm is based on the Thomas algorithm [2], which solves tridiagonal systems. Although the use of GPUs to compute the Thomas algorithm has been deeply studied [17, 20, 5, 22, 3], the differences among these two algorithms, Hines and Thomas, makes us impossible to use the last one as this can not deal with the sparsity of the Hines matrix.

Previous works [1] have explored the use of other algorithms based on the Stone's method [11]. Unlike Thomas algorithm, this method is parallel. However, it is in need of a higher number of operations ($20n \log 2n$) with respect to the ($8n$) operations of the Thomas algorithm to solve one single system of size n. Also, the use of parallel methods present some additional drawbacks to be dealt with. For instance, it would be difficult to compute those neurons that compromise a size bigger than the maximum number of threads per CUDA block (1024) or shared memory (48KB in NVIDIA k80 and 64KB in NVIDIA P100). Other problems are the computationally expensive operations such as atomic accesses and synchronizations necessary to compute this method. Each neuron

presents a particular morphology and so a different scheduling (preprocessing) must be applied to each of them which makes even more difficult its implementation.

Unlike the work presented in [1], where a relatively low number of neurons (128) is computed using single precision operations, in this work we are able to execute a very high number of neurons (up to hundreds of thousands) using double precision operations. We have used the Hines algorithm, which is the optimum method in terms of number of operations, avoiding high expensive computational operations, such as synchronizations and atomic accesses. Our code is able to compute a high number of systems (neurons) of any size in one call (CUDA kernel), using one thread per Hines system instead of one CUDA block per system. Although multiple works have explore the use of GPUs to compute multiple independent problems in parallel without transforming the data layout [13, 14, 12, 18], the particular characteristics of the sparsity of the Hines matrices forces us to modify the data layout to efficiently exploit the memory hierarchy of the GPUs (coalescing accesses to GPU memory). These modifications have not been explored previously, which are deeply described and analyzed in the present work.

1.2 Motivation and Objectives

The broad objective of this thesis is the optimization of Arbor in order to exploit better the new many-core architectures, concretely GPU architectures.

Through a quick analysis of the simulation we determined, as we mentioned before, that the voltage computation was the most time consuming step, leading us to shift our focus to implement a GPU kernel able to solve that tasks at least twice faster.

As we will see in Section 4.1, our implementation obtain the desired 2x speed-up over the multi-core implementation or even more depending of the test case.

Chapter 2

State of the art

2.1 Graphics Processing Unit (GPU)

Although GPUs are traditionally associated to interactive applications involving high rasterization performance, they are also widely used to accelerate much more general applications (now called General Purpose Computing on GPU (GPGPU) [106]) which require an intense computational load and present parallel characteristics. The main feature of these devices is a large number of processing elements integrated into a single chip, which reduces significantly the cache memory. These processing elements can access to a local high-speed external DRAM memory, connected to the computer through a high-speed I/O interface (PCI-Express). Overall, these devices can offer a higher main memory bandwidth and can use data parallelism to achieve a higher floating point throughput than CPUs [44]. Figure 2.1 (top) describes the architecture of modern NVIDIA's GPUs. It consists of a number of multiprocessors and each multiprocessor has a set of simple cores. All multiprocessors share the same main memory, called "global memory". In addition, all cores of one multiprocessor can access to the same "shared memory". This memory is useful when many threads have to access to the same data or if one data is used many times by one thread. Indeed when a block of information has to be loaded in shared memory it is necessary to take it from global memory. To control the GPU devices and manage memory, we have used in the present work the high level programming language CUDA [28], introduced by NVIDIA.

Calculations in CUDA are distributed into a mesh or grid of thread blocks of the same size (number of threads). These threads run the GPU code, known as kernel; note that although this kernel is originally called by the CPU, finally it is executed in the GPU, as seen in Figure 2.1(bottom). Threads within a blocks are grouped into warps of 32 threads. A warp executes one common instruction at a time, so full efficiency is done when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths. The dimensions of both the mesh and the threads block should be carefully chosen in order to achieve the maximum performance depending on the specific problem being treated.

The threads within a block can work together efficiently exchanging data via a

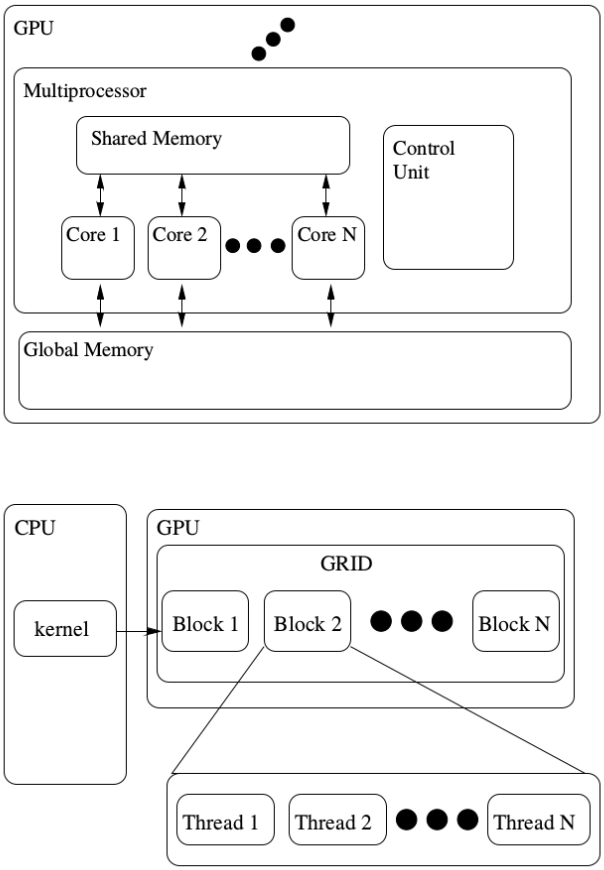

Figure 2.1: GPU architecture (top) and grid of CUDA blocks (bottom).

local shared memory and synchronize low-latency execution through synchronization barriers (where threads in a block are suspended until they all reach the synchronization point). By contrast, the threads of different blocks in the same network can only communicate through a high-latency access to global memory (the memory graphic board). In order to exploit the bandwidth of both global and shared memory in an efficient way, it is better that threads have the same or very similar pattern of memory access to reach contiguous spaces of memory (coalescing access). Besides, another technique used to avoid the latency of the global memory, consists in overlapping the executions of threads blocks with accesses to global memory.

All CUDA code is divided in two different parts, CPU code and GPU code (kernel). The CPU code, provides the instructions to be performed by the CPU, e.g. allocating data on the CPU and GPU, transferring data between GPU and CPU and launching kernels. On the other hand, the GPU code provides the instructions to be executed in the GPU, by all threads of the kernel.

2.1.1 cuSPARSE

The cuSPARSE[1] library contains a set of basic linear algebra subroutines used for handling sparse matrices. It is implemented on top of the NVIDIA CUDA runtime

[1] `http://docs.nvidia.com/cuda/cusparse/index.html`

(which is part of the CUDA Toolkit) and is designed to be called from C and C++. The library routines can be classified into four categories:

- *Level 1*: operations between a vector in sparse format and a vector in dense format
- *Level 2*: operations between a matrix in sparse format and a vector in dense format
- *Level 3*: operations between a matrix in sparse format and a set of vectors in dense format (which can also usually be viewed as a dense tall matrix)
- *Conversions*: operations that allow conversion between different matrix formats, and compression of csr matrices.

The cuSPARSE library allows developers to access the computational resources of the NVIDIA graphics processing unit (GPU), although it does not auto-parallelize across multiple GPUs. As we will see in subsequent sections this library contains the reference solver for tridiagonal linear systems in GPU, *gtsvStridedBatch* which is going to be a great reference in order to evaluate our implementations.

2.2 Tridiagonal Linear Systems

The state-of-the-art method to solve tridiagonal systems is the called Thomas algorithm [15]. Thomas algorithm is a specialized application of the Gaussian elimination that takes into account the tridiagonal structure of the system. Thomas algorithm consists of two stages, commonly denoted as forward elimination and backward substitution.

Given a linear $Au = y$ system, where A is a tridiagonal matrix:

$$A = \begin{bmatrix} b_1 & c_1 & & & & 0 \\ a_2 & b_2 & c_2 & & & \\ & \cdot & \cdot & \cdot & & \\ & & \cdot & \cdot & \cdot & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{bmatrix}$$

The forward stage eliminates the lower diagonal as follows:

$$c'_1 = \frac{c_1}{b_1}, \quad c'_i = \frac{c_i}{b_i - c'_{i-1} a_i} \quad \text{for } i = 2, 3, \ldots, n-1$$
$$y'_1 = \frac{y_1}{b_1}, \quad y'_i = \frac{y_i - y'_{i-1} a_i}{b_i - c'_{i-1} a_i} \quad \text{for } i = 2, 3, \ldots, n-1$$

and then the backward stage recursively solve each row in reverse order:

$$u_n = y'_n, u_i = y'_i - c'_i u_{i+1} \text{ for } i = n-1, n-2, \ldots, 1$$

Overall, the complexity of Thomas algorithm is optimal: $8n$ operations in $2n - 1$ steps.

Cyclic Reduction (CR) [21, 8, 15, 16] is a parallel alternative to Thomas algorithm. It also consists of two phases (reduction and substitution). In each intermediate step of the reduction phase, all even-indexed (i) equations $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$ are reduced. The values of a_i, b_i, c_i and d_i are updated in each step according to:

$$a'_i = -a_{i-1}k_1, b'_i = b_i - c_{i-1}k_1 - a_{i+1}k_2 \; c'_i = -c_{i+1}k_2, y'_i = y_i - y_{i-1}k_1 - y_{i+1}k_2$$
$$k_1 = \frac{a_i}{b_{i-1}}, k_2 = \frac{c_i}{b_{i+1}}$$

After $\log_2 n$ steps, the system is reduced to a single equation that is solved directly. All odd-indexed unknowns x_i are then solved in the substitution phase by introducing the already computed u_{i-1} and u_{i+1} values:

$$u_i = \frac{y'_i - a'_i x_{i-1} - c'_i x_{i+1}}{b'_i}$$

Overall, the CR algorithm needs $17n$ operations and $2\log_2 n - 1$ steps. Figure 2.2 graphically illustrates its access pattern.

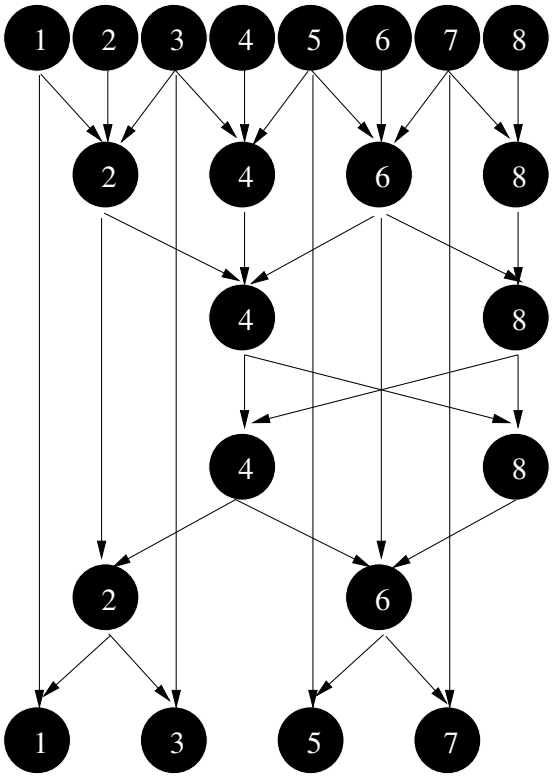

Figure 2.2: Access pattern of the CR algorithm.

Parallel Cyclic Reduction (PCR) [21, 8, 15, 16] is a variant of CR, which only has substitution phase. For convenience, we consider cases where $n = 2^s$, that involve $s = \log_2 n$ steps. Similarly to CR a, b, c and y are updated as follows, for $j = 1, 2, \ldots, s$ and $k = 2^{j-1}$:

$$a'_i = \alpha_i a_i, b'_i = b_i + \alpha_i c_{i-k} + \beta_i a_{i+k}$$
$$c'_i = \beta_i c_{i+1}, y'_i = b_i + \alpha_i y_{i-k} + \beta_i y_{i+k}$$
$$\alpha_i = \frac{-a_i}{b_{i-1}}, \beta_i = \frac{-c_i}{b_i}$$

finally the solution is achieved as:

$$u_i = \frac{y'_i}{b_i}$$

Essentially, at each reduction stage, the current system is transformed into two smaller systems and after $\log_2 n$ steps the original system is reduced to n independent equations. Overall, the operation count of PCR is $12n \log_2 n$. Figure 2.3 sketches the corresponding access pattern.

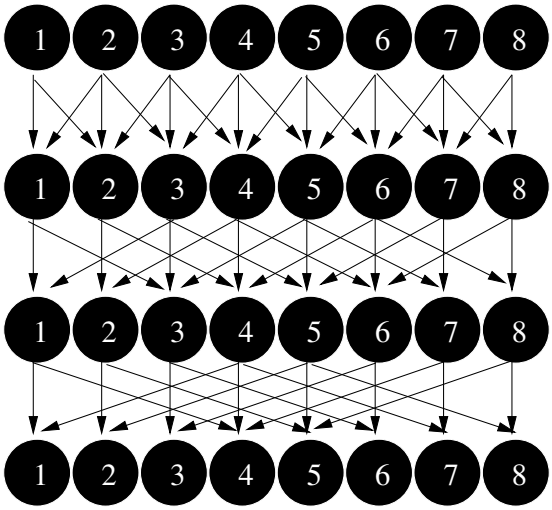

Figure 2.3: Access pattern of the PCR algorithm.

We should highlight that apart from their computational complexity these algorithms differ in their data access and synchronization patterns, which also have a strong influence on their actual performance. For instance, in the CR algorithm synchronizations are introduced at the end of each step and its corresponding memory access pattern may cause bank conflicts. PCR needs less steps and its memory access pattern is more regular [21].

In fact, hybrid combinations that try to exploit the best of each algorithm have been explored [21, 10, 4, 8, 15, 16]. Figure 2.4 illustrates the access pattern of the CR-PCR combination proposed in [21]. CR-PCR reduces the system to a certain size using the forward reduction phase of CR and then solves the reduced (intermediate) system with the PCR algorithm. Finally, it substitutes the solved unknowns back into the original system using the backward substitution phase of CR. Indeed, this is the method implemented by the *gtsvStridedBatch* routine into the *cuSPARSE* package [3].

There are more algorithms apart of the ones above mentioned to deal with tridiagonal systems, such as those based on Recursive Doubling [21], among others. However we have focused on those, which were proven to achieve a better performance and were implemented in the reference library [3].

2.2.1 Hines Algorithm

In this section, we describe the numerical framework behind the computation of the Voltage on neurons morphology. It follows the next general form:

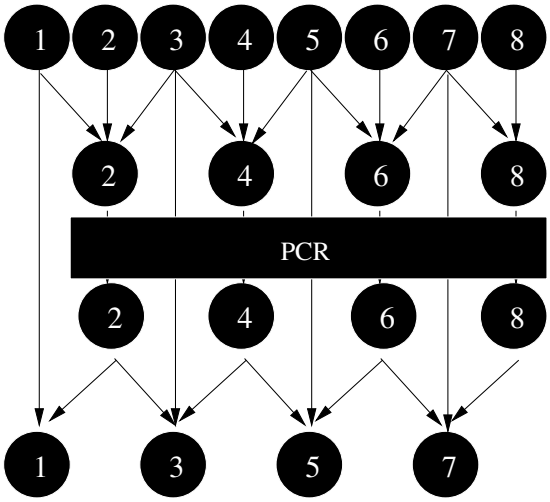

Figure 2.4: Access pattern of the CR-PCR algorithm.

$$C\frac{\partial V}{\partial t} + I = f\frac{\partial}{\partial x}(g\frac{\partial V}{\partial x}) \tag{2.1}$$

where f and g are functions on x-dimension and the current I and capacitance C [6] depend on the voltage V. Discretizing the previous equation on a given morphology we obtain a system that has to be solved every time-step. This system must be solved at each point:

$$a_i V_{i+1}^{n+1} + d_i V_i^{n+1} + b_i V_{i-1}^{n+1} = r_i \tag{2.2}$$

where the coefficients of the matrix are defined as follow:

$$\text{upper diagonal: } a_i = -\frac{f_i g_{i+\frac{1}{2}}}{2\Delta_x^2}$$
$$\text{lower diagonal: } b_i = -\frac{f_i g_{i+\frac{1}{2}}}{2\Delta_x^2}$$
$$\text{diagonal: } d_i = \frac{C_i}{\Delta_t} - (a_i + b_i)$$
$$\text{rhs: } r_i = \frac{C_i}{\Delta_t} V_i^n - I - a_i(V_{i-1}^n - V_i^n) - b_i(V_{i+1}^n - V_i^n)$$

The a_i and b_i are constant in the time, and they are computed once at start up. Otherwise, the diagonal (d) and right-side-hand (rhs) coefficients are updated every time-step when solving the system.

The discretization above explained is extended to include *branching*, where the spatial domain (neuron morphology) is composed of a series of one-dimension *sections* that are joined at branch points according to the neuron morphology.

For sake of clarity, we illustrate a simple example of a neuron morphology in Figure 2.5. It is important to note that the graph formed by the neuron morphology is an acyclic graph, i.e. it has no loops. The nodes are numbered using a scheme that gives the matrix sparsity structure that allows to solve the system in linear time.

To describe the sparsity of the matrix from the numbering used, we need an array $(p_i \; i \in [2:n])$ which stores the parent indexes of each node. The pattern of the matrix which illustrates the morphology shown above is graphically illustrated in Figure 2.5.

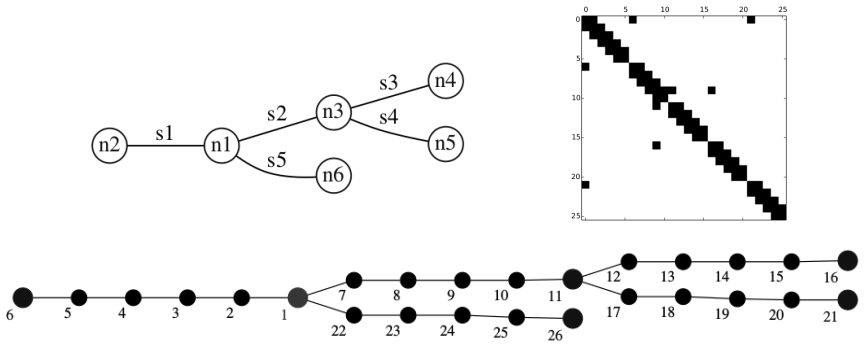

Figure 2.5: Example of a neuron morphology and its numbering (left-top and bottom) and sparsity pattern corresponding to the numbering followed (top-right).

The Hines matrices feature the following properties: they are symmetric, the diagonal coefficients are all nonzero and per each off-diagonal element, there is one off-diagonal element in the corresponding row and column (see row/column 7, 12, 17 and 22 in Figure 2.5).

Given the aforementioned properties, the Hines systems ($Ax = b$) can be efficiently solved by using an algorithm similar to Thomas algorithm for solving tri-diagonal systems. This algorithm, called Hines algorithm, is almost identical to the Thomas algorithm except by the sparsity pattern given by the morphology of the neurons whose pattern is stored by the p vector. An example of the sequential code used to implement the Hines algorithm is illustrated in pseudo-code in Algorithm 1.

Algorithm 1 Hines algorithm.

```
 1: void solveHines(double *u, double *l,   double *d,
 2:                 double *rhs,      int *p, int cellSize)
 3: // u → upper vector, l → lower vector
 4: int i;
 5: double factor;
 6: // Backward Sweep
 7: for i = cellSize − 1 → 0 do
 8:     factor    = u[i] / d[i];
 9:     d[p[i]]   -= factor × l[i];
10:     rhs[p[i]] -= factor × rhs[i];
11: end for
12: rhs[0] /= d[0];
13: // Forward Sweep
14: for i = 1 → cellSize − 1 do
15:     rhs[i] -= l[i] × rhs[p[i]];
16:     rhs[i] /= d[i];
17: end for
```

Chapter 3

Design and Development

3.1 Implementation of cuThomasBatch

An efficient memory management is critical to achieve a good performance, but even much more on those architectures based on a high throughput and a high memory latency, such as the GPUs. In this sense, first we focus on presenting the different data layouts proposed and analyze the impact of these on the overall performance. Two different data layouts were explored: *Flat* and *Full-Interleaved*. While the *Flat* data layout consists of storing all the elements of each of the systems in contiguous memory locations, in the *Full-Interleaved* data layout, first, we store the first elements of each of the systems in contiguous memory locations, after that we store the set of the second elements, and so on until the last element.

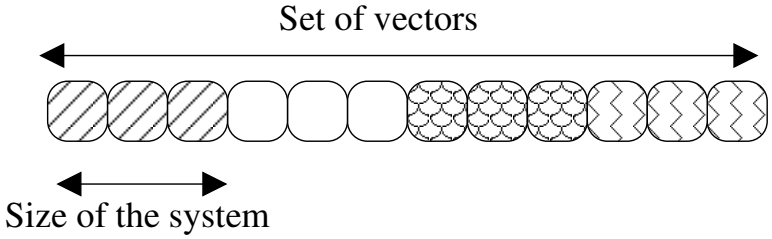

Figure 3.1: Example of the *Flat* data layout.

For sake of clarity, Figure 3.1 and 3.2 illustrate a simple example composed by four different tridiagonal systems of three elements each. Please, note that we only illustrate one vector per system in Figure 3.1, but in the real scenario we would have 4 vectors per tridiagonal system on which are carried out the strategies above described. As widely known, one of the most important requirements to achieve a good performance on NVIDIA GPUs is to have contiguous threads accessing contiguous memory locations (coalescing memory accesses). This is the main motivation behind the proposal of the different data layouts and CUDA thread mappings. As later shown, the differences found in the data layouts studied have important consequences on the scalability.

Additionally, we have explored other data-layout, *Unified-Vector*. In this case, we attempt to analyze the hierarchy of memory by exploiting the temporal locality that there is among the different vectors (a, b, c, and u/y in Section 2.2). Basically, every thread, immediately after a_i is computed, has to compute also b_i and c_i, in the forward step. In the backward step, the process is similar, but in the opposite order. Using this

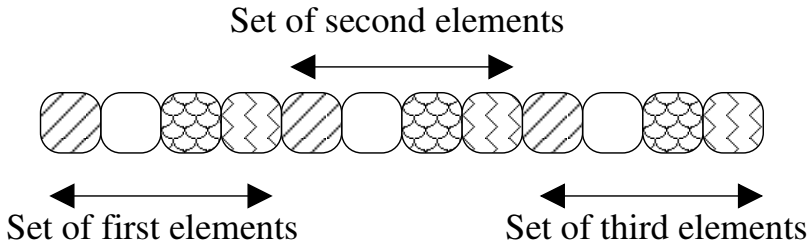

Figure 3.2: Example of the *Full-Interleaved* data layout.

data layout, we want to take advantage of this characteristic of the *Thomas* algorithm. For sake of clarity, Figure 3.3 graphically illustrates the data layout proposed for a simple batch composed by only 2 independent tridiagonal systems.

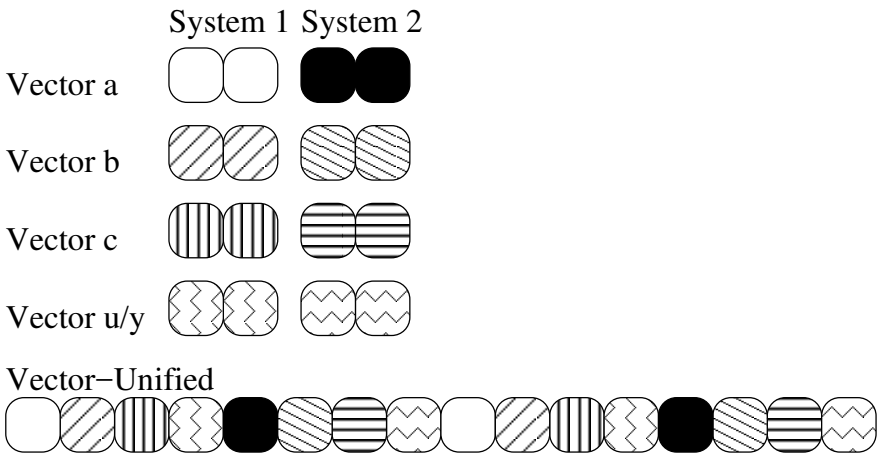

Figure 3.3: Example of the *Unified-Vector* data layout.

Next, we explore the different proposals about the CUDA thread mapping on the data layouts above described. Figure 3.4 illustrates the different CUDA thread mappings studied in this paper. Figure 3.4(top) shows a coarse-grain scheme where a set of tridiagonal ($S_1, .., S_n$ in Figure 3.4) systems is mapped onto a CUDA block, so that each CUDA thread fully solves one system. We decided to explore this approach to avoid dealing with atomic accesses and synchronizations, as well as to be able to execute a very high number of tridiagonal systems of any size, without the limitation imposed by the parallel methods.

Using the *Flat* data layout we can not exploit coalescence when exploiting one thread per tridiagonal system (coarse approach) [15]; however by interleaving (Figure 3.2) the elements of the vectors (a, b, c, u and y in Section 2.2), contiguous threads access to contiguous memory locations. This approach does not exploit efficiently the shared memory of the GPUs, since the memory required by each CUDA thread becomes too large. Our GPU implementation (*cuThomasBatch*) is based on this approach, *Thomas* algorithm (Section 2.2) on *Full-Interleaved* data layout (Figure 3.2).

On the other hand, previous studies have explored the use of the fine-grain scheme based on CR-PCR [21, 8, 15, 16] using the *Flat* data layout. In this case (Figure 3.4(bottom)), each tridiagonal system is distributed across the threads of a CUDA

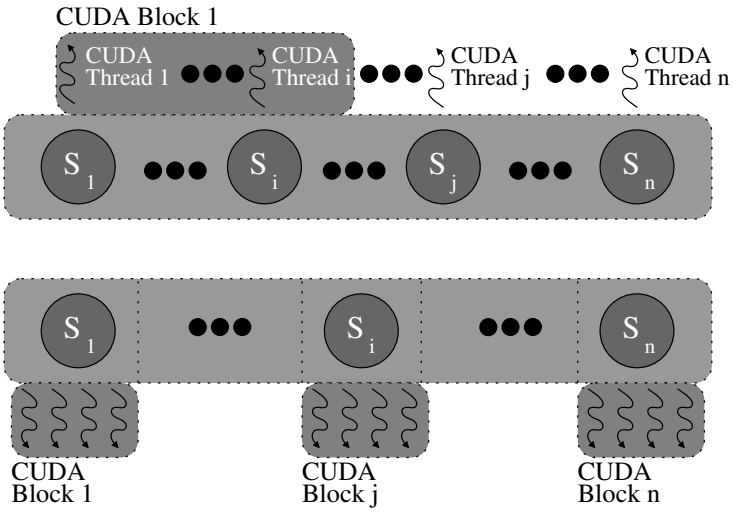

Figure 3.4: Coarse (top) and fine (bottom) CUDA thread mapping.

block so that the shared memory of the GPU can be used more effectively (both the matrix coefficients and the right hand side of each tridiagonal system are hold on the shared memory of the GPU). However, computationally expensive operations, such as synchronizations and atomic accesses are necessary. Also this approach saturates the capacity of the GPU with a relatively low number of tridiagonal systems. Although the shared memory is much faster than the global memory, it presents some important constraints to deal with. This memory is useful when the same data can be reused either by the same thread or by other thread of the same block of threads (CUDA block). Also, it is small (up to 64KB in the architecture used [9]) and its use hinders the exchange among blocks of threads by the CUDA scheduler to overlap accesses to global memory with computation. Our reference implementation (the *gtsvStridedBatch* routine into the cuSPARSE package [3]) is based of this approach, CR-PCR (Section 2.2) on *Flat* data layout (Figure 3.1).

3.2 Implementation of cuHinesBatch

As we mentioned previous in this work, we can say that Hines solver is a particular case of Thomas solver, for this reason all of our proposed approaches are based on the previous knowledge.

As in cuThomasBatch implementation, we are going to explore different data layouts: *Flat* and *Full-Interleaved*, explained in the previous section, and *Block-Interleaved*, that similarly to the *Full-Interleaved* data layout, the *Block-Interleaved* data layout divides the set of systems in groups of systems of a given size (BS), whose elements are stored in memory by using the strategy followed by the *Full-Interleaved* approach.

Same as in Figure 3.1, please note that we only illustrate one vector per system in Figure 3.5, but in the real scenario we would have 4 vectors per Hines system.

Although, we exploit coalescence in the memory accesses by using the *Full-Interleaved* approach, the threads have to jump in memory as many elements as the number of systems to access the next element of the vector(s) (Point-lines in Figure 3.5). This could cause an inefficient use of the memory hierarchy. This is why we study an additional

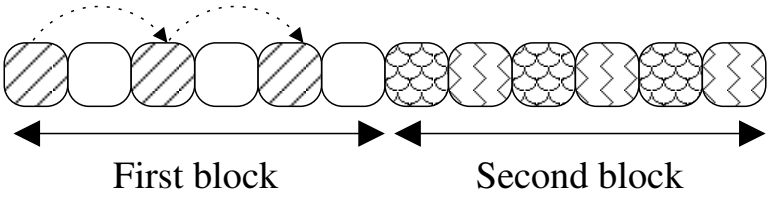

Figure 3.5: Example of *Block-Interleaved* data layout with a *BS* equal to 2, for 4 Hines systems of three elements each. Point-line represents the jumps in memory carried out by the first thread/system.

approach, the called *Block-Interleaved* data layout. Using this approach we reduce the number of elements among consecutive elements of the same system, and so the jumps in memory are not as big as in the previous approach (*Full-Interleaved*), while keeping the coalesced memory accesses. Also, the use of the *Block-Interleaved* data layout can take advantage better of the growing importance of the bigger and bigger cache memories in the memory hierarchy of the current and upcoming CUDA architectures.

3.2.1 Implementation based on Shared Memory

Unlike the previous approaches, here we explore the use of shared memory for our target application. The shared memory is much faster than the global memory; however it presents some important constraints to deal with. This memory is useful when the same data can be reused either by the same thread or by other thread of the same block of threads (CUDA block). Also, it is small (until 48KB in the architecture used) and its use hinders the exchange among blocks of threads by the CUDA scheduler to overlap accesses to global memory with computation.

As we can see in Pseudocode 1, in our problem the elements of the vectors a, d, b, rhs and p are reused in the *Forward Sweep* after computing the *Backward Sweep*. However, the granularity used (1 thread per system) and the limit of the shared memory (48KB) prevents from storing all the vectors in shared memory. To be able to use shared memory we have to use the *Block-Interleaved* data layout. The number of systems to be grouped (*BS*) is imposed by the size of the shared memory. In order to address the limitation of shared memory, we only store the *rhs* vector, as this is the vector on which more accesses are carried out. In this sense, the more systems are packed in shared memory, the more accesses to shared memory are carried out.

3.3 Variable Batch

In this section, we describe the techniques used to deal with Variable Batch, batch of tridiagonal(Thomas and Hines) systems with different sizes. Figure 3.6 graphically illustrates a simple example of Variable Batch composed by three vectors of different size. To deal with Variable Batch, we make use of a widely used and extended technique very popular in parallel programming, the so called padding. This technique basically consists of filling with null elements those memory locations between two different data of different size. This is particularly interesting and beneficial for GPU-based architectures, where the pattern of access to memory is critical to achieve coalescing and reduce the impact of the high latency. However, in our particular scenario, we have to adapt this technique to the data-layout used, *Full-Interleaved*. An example of this

is illustrated by Figure 3.6.

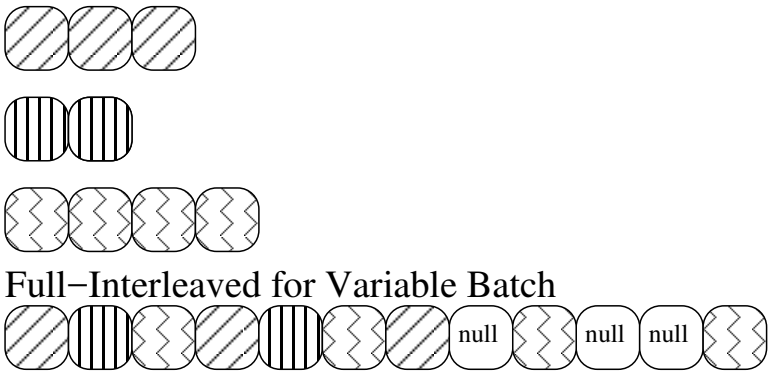

Figure 3.6: Example of the *Full-Interleaved* data layout for Variable Batch (Padding).

Regarding the CUDA thread mapping, we follow the approach based on using one CUDA thread per tridiagonal thread (Figure 3.4-top). The reference code (*gtsvStrided-Batch*) does not offer the possibility to compute batch of tridiagonal system of different size. We have evaluated two different approaches, one called *Computing Padding* (CP) and one called *No Computing Padding* (NCP). While the *CP* approach computes the null elements located between different inputs, which does not affect to the final result, the *NCP* only computes the no-null elements. This last can be shown as a more efficient approach, but it is in need of a extra parameter (vector) which stores the size of the systems and it suffers from divergence. Those threads that are in charge of computing small systems, stop before others which have to compute large systems. This provokes not only divergence among different threads in the same CUDA block, but also no coalesce memory accesses, which can affect to performance considerably.

Chapter 4

Performance Evaluation

The following chapter presents the evaluations performed throughout the project, including the analysis of the different proposed approaches for both presented implementations, *cuThomasBatch* and *cuHinesBatch*.

4.1 Evaluation Environment

In this section the technical information about the used platforms is provided, together with all the environment setup in order to bring to the reader the possibility of reproduce the results by himself.

4.1.1 CTE-POWER

CTE-POWER is a cluster based on IBM Power8+ processors (Power Minsky), with a Linux Operating System and an Infiniband interconnection network.

- 6 computing nodes, each with the following technical characteristics:
  - 2x IBM PowerNV 8335-GTB @ 4.00GHz (10 cores and 8 threads/core, total 160 threads per node)
  - 256 GB of main memory distributed in 32 dimms x 8GB @ 2400MHz
  - 2x 480GB SSD as local storage
  - 1.6TB NVMe
  - 2x nVidia Pascal P100 GPU with 16GB of memory.
  - Dual Port Mellanox EDR
  - GPFS via two fiber links 10 GBit

The operating system is Red Hat Enterprise Linux Server 7.3 (Maipo). This platform was use to carry out all the experiments based on cuThomasBatch, We have used the next configuration (compilers version and flags): gcc 4.8.5, cuda 8.0, -arch=sm_60 -fopenmp -O3 -std=c++11

4.1.2 Minotauro

MinoTauro is a heterogeneous cluster with 2 configurations:

- 61 Bull B505 blades, each blade with the following technical characteristics:
  - 2 Intel E5649 (6-Core) processor at 2.53 GHz
  - 2 M2090 NVIDIA GPU Cards
  - 24 GB of Main memory
  - Peak Performance: 88.60 TFlops
  - 250 GB SSD (Solid State Disk) as local storage
  - 2 Infiniband QDR (40 Gbit each) to a non-blocking network
  - 14 links of 10 GbitEth to connect to BSC GPFS Storage
- 39 bullx R421-E4 servers, each server with:
  - 2 Intel Xeon E5–2630 v3 (Haswell) 8-core processors, (each core at 2.4 GHz,and with 20MB L3 cache)
  - 2 K80 NVIDIA GPU Cards
  - 128 GB of Main memory, distributed in 8 DIMMs of 16 GB – DDR4 @ 2133 MHz - ECC - SDRAM
  - Peak Performance: 250.94 TFlops
  - 120 GB SSD (Solid State Disk) as local storage
  - 1 PCIe 3.0 x8 8GT/s, Mellanox ConnectXR–3FDR 56 Gbit
  - 4 Gigabit Ethernet ports.

The operating system is RedHat Linux 6.7 for both configurations. This platform was used to carry out the cuHinesBatch experiments. The main reason in order to use Minotauro and not CTE-POWER despite being newer, is the lack of enough GPU device in order to carry out the multi-GPU experiments. We have used the next configuration (compilers version and flags): gcc 4.4.7, nvcc (CUDA) 7.5, -O3, -fopenmp, -arch=sm_37.

4.2 cuThomasBatch Performance Analysis

We have evaluated the performance of each of the approaches, *gtsvStridedBatch*, *cuThomasBatch*, *cuThomasBatch-Unified Vector* using both, single and double precision operations. Our test cases consist of computing 256, 2,560, 25,600 and 256,000 tridiagonal systems of 64, 128, 256 and 512 elements each. We have considered this test bed to evaluate the scalability by increasing both, the size of the systems and the number of systems, taking into account the limitation of our platform. In particular the size of the systems in the test cases (64-512) can be fully executed by one CUDA block using *gtsvStridedBatch*. Regarding the size of the tridiagonal systems, there is no a characteristic size, as it depends on the nature of the applications, and because of that, we have considered different cases to cover all the range of possible scenarios.

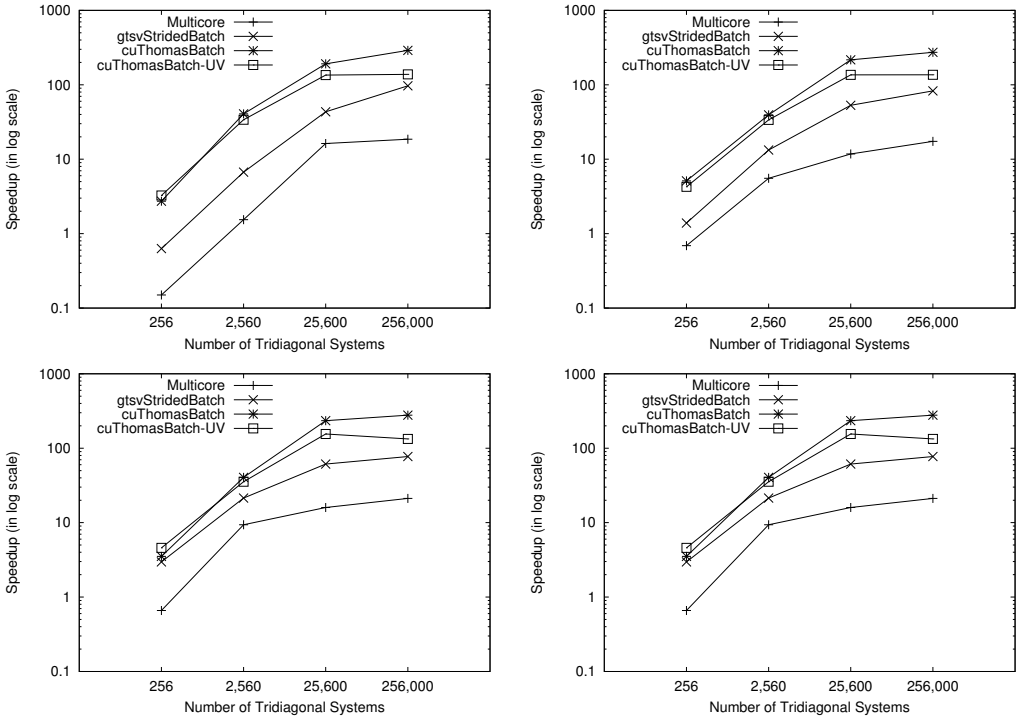

Figure 4.1: Speedup (execution time of each of the approaches divided by the execution time of the sequential CPU code) of *Multicore(Multi)*, *cuThomasBatch*, *gtsvStrided-Batch* and *cuThomasBatch-UnifiedVector(UV)* using **single precision** operations for computing multiple, 256-256,000 tridiagonal systems using different sizes: 64 (left-top), 128 (right-top), 256 (left-bottom) and 512 (right-bottom).

4.2.1 Scalability

To analyze in detail the scalability of all the implementations, we graphically illustrate in Figures 4.1 and 4.2 the speedup for both, single and double precision operations, against the sequential counterpart including the performance achieved by the multicore execution (20 cores in 2 sockets IBM Power8 node, 10 cores each). The implementation based on multicore basically makes use of an OpenMP pragma (*#pragma omp for*) on the top of the for loop which goes over the different independent tridiagonal systems to distribute blocks of systems over the available cores. While *gtsvStridedBatch* achieves a peak speedup about 90 in single precision and closed to 80 in double precision, *cuThomasBatch* scales much more (Figure 4.1 and Figure 4.2) achieving a speedup peak about 280 in single precision and about 180 for double precision operations. It is remarkable that the use of different precisions has a higher impact in *cuThomasbatch* than in *gtsvStridedBatch*. It is important to note that in some cases, the multicore OpenMP implementation outperforms, in some cases (25,600 and 256,000 systems), the *gtsvStridedBatch* implementation when executing systems of small size (64). Regarding the *cuThomasBatch-Unified Vector*, this only outperforms the *cuThomasBatch* performance in a few cases (256 systems of size 64, for instance). The scalability of this approach is considerably worse, as it suffers from a fall in performance when a high number of systems is executed.

Figure 4.3 graphically illustrates the speedup achieved by our implementation (*cuThomas-*

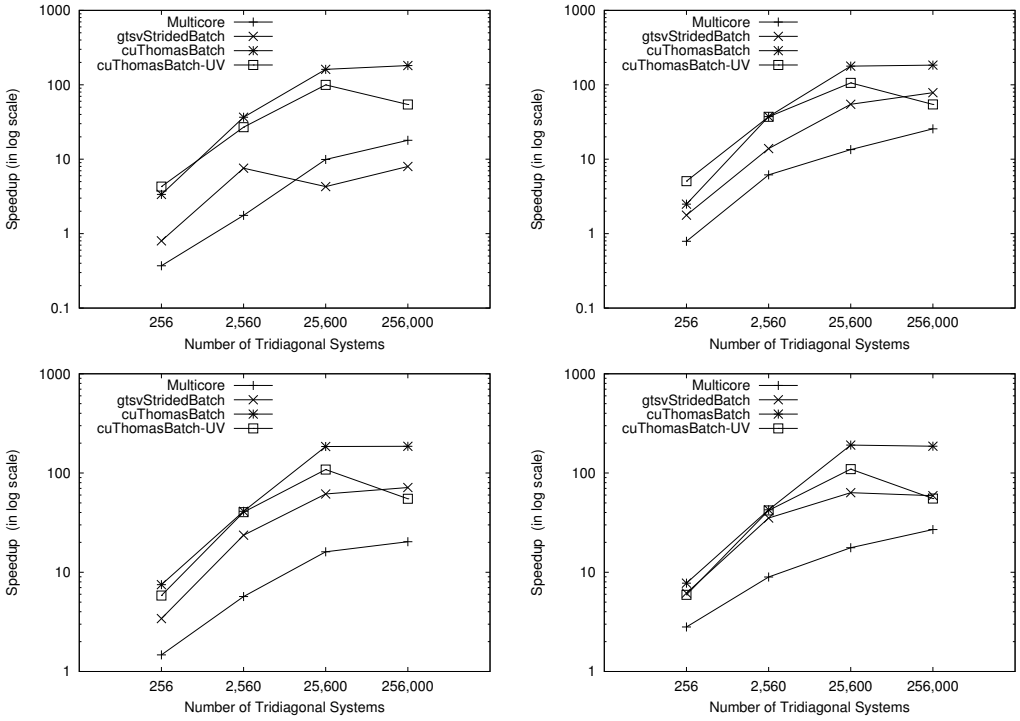

Figure 4.2: Speedup (execution time of each of the approaches divided by the execution time of the sequential CPU code) of *Multicore(Multi)*, *cuThomasBatch*, *gtsvStrided-Batch* and *cuThomasBatch-UnifiedVector(UV)* using **double precision** operations for computing multiple, 256-256,000 tridiagonal systems using different sizes: 64 (left-top), 128 (right-top), 256 (left-bottom) and 512 (right-bottom).

Batch) against the *cuSPARSE* routine. There is no a clear trend regarding the benefit achieved using *cuThomasBatch*, as it depends on many different factors. However, this implementation is always better than the *gtsvStridedBatch* routine, obtaining a speedup range from 1.25 to 6.1 in single precision and from 1.21 to close to 5 for double precision operations.

To evaluate both implementations, *gtsvStridedBatch* and *cuThomasBatch*, more deeply, we make use of the NVIDIA profiler (NVPROF) to achieve some metrics like bandwidth and efficiency (Warp Execution Efficiency in NVPROF). The peak bandwidth on P100 is 732 GB/s, however the ECC (Error Correcting Code), which is natively supported in the HBM2 memory, causes a fall in the bandwidth about 15% [9]. So the real bandwidth in P100 is about 622 GB/s. To obtain these metrics, we have used the biggest test case, 256,000 systems of 512 elements each. Unlike our implementation, the *gtsvStridedBatch* routine is composed by two kernels, *gtsvFirstPassKernelMEM* and *gtsvSharedMemKernel*. While the first is more focused on global memory operations, the second concentrates more operations on shared memory. As commented, the algorithm used in *gtsvStridedBatch*, see Section 2.2, can exploit efficiently shared memory. The bandwidth achieved by *gtsvFirstPassKernelMEM* is about 503 GB/s (about 80% of the real bandwidth). In *gtsvSharedMemKernel* the bandwidth is mush smaller, about 192 GB/s, however, this kernel focuses on shared memory operations, achieving a bandwidth on this memory about 5,670 GB/s. In *cuThomasBatch* the bandwidth achieved is 527 GB/s. The kernels of the *cuSPARSE* routine, *gtsvFirstPassKernelMEM* and

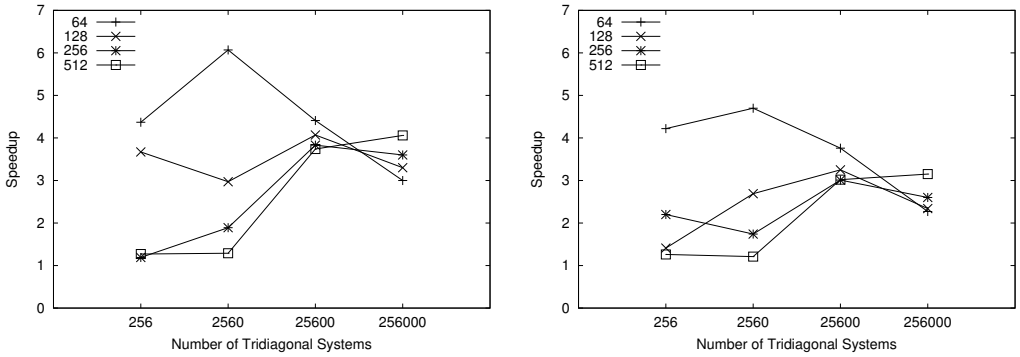

Figure 4.3: Speedup (execution time of the *cuThomasBatch* by the execution time of the *gtsvStridedBatch*) using both, single precision (left) and double precision operations (right), for computing multiple, 256-256,000 tridiagonal systems using different sizes: 64, 128, 256 and 512.

gtsvSharedMemKernel, are able to achieve an efficiency about 99.9% and 94.1% respectively. In our implementation the efficiency is 100%.

It is also important to highlight that *cuThomasBatch*, unlike the *gtsvStridedBatch*, is in need to modify the data layout by interleaving the elements of the vectors. This preprocessing does not compromise an important overhead with respect to the whole process, in those applications (numerical simulations) which have to solve multiple tridiagonal systems many times in a temporal range, as this is carried out just once at the very beginning of the simulation [19], being that the case of the neuronal networks simulators.

4.2.2 Numerical Accuracy

The numerical accuracy is critical in a large number of scientific and engineering applications. In this sense, we compared the numerical accuracy of both parallel approaches against the sequential counterpart, increasing the size of the system. For sake of numerical stability we force the tridiagonal coefficient matrix be diagonally dominant ($|b_i| > |a_i| + |c_i|, \forall i = 0, \ldots, n$). We initialize the matrix coefficients randomly following the previous property. The error (accuracy with respect to the result obtained by the sequential CPU code) in *cuThomasBatch* is zero. This is because of the intrinsic characteristics of the Thomas algorithm. On the other hand, the error using *gtsvStridedBatch* is between 7×10^{-8} and 9×10^{-8} for single precision and between 1×10^{-16} and 2×10^{-16} for double precision on the tests evaluated. One additional important characteristic of our implementation is that the results, in terms of numerical accuracy, are reproducible. This is an important characteristics for multiple applications.

4.2.3 Memory Occupancy

As commented in Section 2.2, the use of parallel methods requires an additional amount of temporary extra storage [3]. In particular *gtsvStridedBatch* is in need of $m \times (4 \times n + 2048) \times sizeof(< type >)$ more memory, being m and n the number of systems and the size of the systems respectively [3]. This supposes, for instance, that *gtsvStridedBatch* requires about 2× more memory capacity than *cuThomasBatch* to compute

256, 000 tridiagonal systems of 64 elements each or about 1GB extra memory to compute 256, 000 tridiagonal systems of 512 elements each (Figure 4.4).

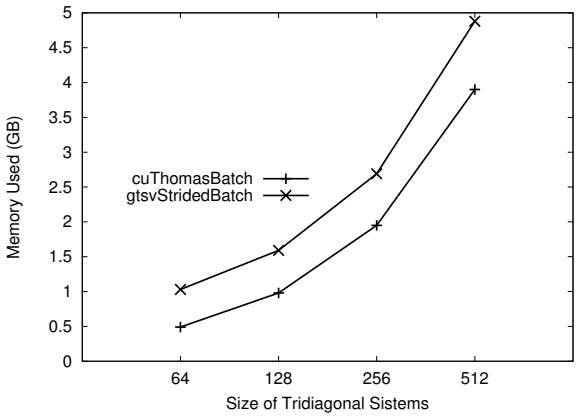

Figure 4.4: Memory used by *gtsvStridedBatch* and *cuThomasBatch* to compute 256,000 tridiagonal systems using double precision.

4.3 cuThomasVBatch

Here we evaluate the variant of *cuThomasBatch* for variable batch (batch of tridiagonal systems with different sizes), *cuThomasVBatch*. To evaluate both variants proposed, *No Computing Padding* (NCP) and *Computing Padding* (CP), we first initialize a batch of tridiagonal systems with a size chosen randomly between 256 and 512. We also compute two other cases to compute batches with the same size, one for 256 and one for 512 using *cuThomasBatch*. As Figure 4.5 illustrates the variant based on *CP* is significantly more efficient and faster than the *NCP* counterpart. This is because of, although the *NCP* needs less number of memory accesses and operations, this variant suffer from divergence and uncoalescing in memory accesses, causing and important underutilization of the computational capacity of our GPU architecture. We also make use of NVPROF to achieve some metrics like bandwidth and efficiency. While the bandwidth achieved by the *NCP* variant is about 205 GB/s, when executing 256,000 tridiagonal systems of 256-512 elements each, the *CP* is able to achieve a bandwidth about 525 GB/s (about 85% of the peak bandwidth) for the same test case. The efficiency is also bigger using the *CP* approach (77%) than the *NCP* one (57%). The performance (execution time) of the *CP* variant is bounded by the biggest size of the batch, as shown in Figure 4.5, the time for a variable batch (*cuThomasVBatch*) of 256-512 is equivalent to the execution time of a fixed-size batch (*cuThomasBatch*) of 512.

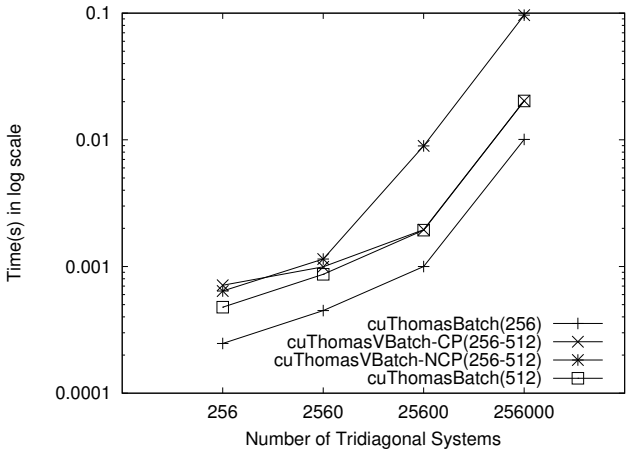

Figure 4.5: Execution time for the two variants, *No Computing Padding* and *Computing Padding*, of the *cuThomasVBatch*.

4.4 cuHinesBatch Performance Analysis

To evaluate the different implementations described in the previous section, we have used real configurations (neurons' morphologies)[1]. In particular, 6 different neurons were used, which can be divided into 6 different categories regarding their sizes and number of branches. More details are described in Table 4.1. We have considered these 6 different morphologies, as a wide range of the neurons fall into the chosen morphologies.

Name	Size	#Branches	Code Name	neuron ID
small-low	76	7	299-DG-IN-Neuron2	NMO_00076
small-high	76	29	202-2-19nj	NMO_00076
medium-low	305	30	59D-40X	NMO_00302
medium-high	319	157	Culture-9-5	NMO_00319
big-low	695	66	28-2-2	NMO_00695
big-high	691	341	HSE-fluoro02	NMO_00691

Table 4.1: Summary of the neurons used.

In this section, 5 different implementations are analyzed. Three of them are the same implementations analyzed in Section 4.2: *Multicore*, *Flat* and *Full-Interleaved* Additionally, we study a *Block-Interleaved* approach explained in Section 3.2 and the use of shared memory (*Block-Shared*) over *Block-Interleaved*. There are multiple different configurations regarding the block-size (*BS*) and CUDA block for the last two scenarios (*Block-Interleaved* and *Block-Shared*). For sake of clarity we focus on one of the possible test cases to evaluate these two approaches. The benefit shown for these two implementations is similar to the rest of test-cases.

First, we evaluate the *Multicore*, *Flat* and *Full-Interleaved* for 256, 2,560, 25,600 and 256,000 *medium-high* (Table 4.1) neurons (Fig. 4.6). On those test cases that do not compromise a high number of neurons (256 and 2,560), *Multicore* obtains better

[1] `http://www.neuromorpho.org/`

performance than the GPU-based implementations. This is mainly because of the parallelism of these tests, which is not enough to saturate GPU and this can not reduce the impact of the high latency by overlapping execution and memory accesses. The use of multicore (16 cores and 2 sockets) supposes a speedup (over sequential execution) about 2 for 256 neurons and about 6 for 256,000 neurons. As shown, *Flat* is not able to scale, even on those test-cases that involve a high number of neurons, being even slower than multicore execution, achieving a maximum speedup about 2. This is because of the memory access pattern which can not exploit coalescing (contiguous threads access to contiguous memory locations). On the other hand, *Full-Interleaved* turns up as the best choice, being faster than *Multicore* and *Flat*, when dealing with a high number of neurons (25,600 and 256,000). Unlike *Flat*, *Full-Interleaved* takes advantage of coalescing when accessing to global memory. As expected, this has an impressive impact on performance, being *Full-Interleaved* about 20× and 25× faster than sequential code when computing 25,600 and 256,000 neurons respectively on one K80 GPU. The use of multiple GPUs is only beneficial on those test cases with an enough computational load where a high number of neurons must be computed (25,600 and 256,000 neurons), with an extra benefit close to the ideal scaling (about 1.9× faster than using one K80 GPU).

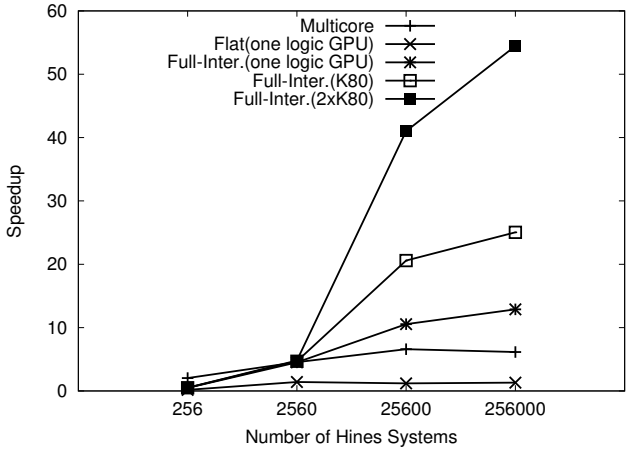

Figure 4.6: Performance (speedup over sequential execution) achieved by multicore (16 cores, 2 sockets) and the GPU-based approaches, *Flat* and *Full-Interleaved* (using different number of GPUs), using medium-high neurons.

As it is not possible to have the control on CUDA scheduler, we have explored a high number of different combinations regarding block-size (*BS*) for the *Block-Interleaved* approach (Section 3.2). For sake of clarity, and given the huge number of different test-cases possible, we have focused on one particular scenario. It consists of computing 256,000 medium-high neurons using different block sizes (*BS*) and fixing the size of the CUDA block (number of threads per block). This is a characteristic case among the tests carried out, as the features (sizes and number of branches) of the morphology used is in between of the other two morphologies. As shown in Figure 4.7(left), some of the cases are slightly better than the *Full-Interleaved* approach, being about a 2% faster.

Next we analyze the performance of the *Block-Shared* implementation. We focus on the same scenario used for the *Block-Interleaved*. Figure 4.7(right) graphically

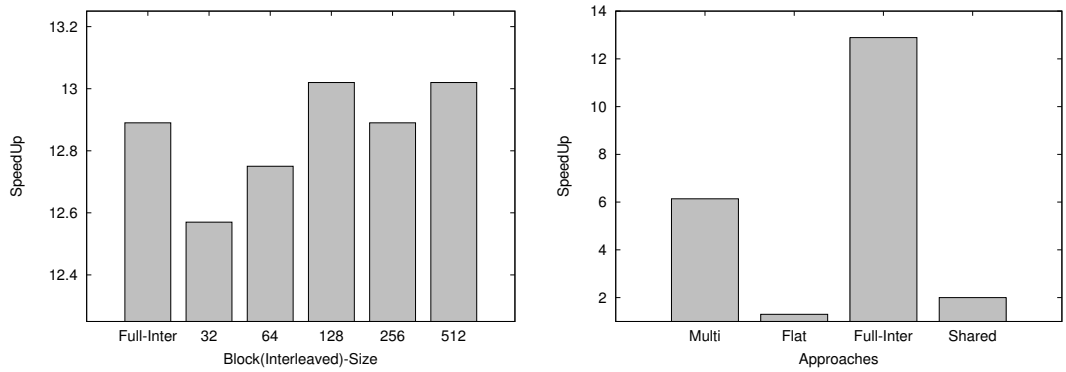

Figure 4.7: (Left) Performance (speedup over sequential execution) achieved by the *Block-Interleaved* approach for multiple *BS* (32, 64, 128, 256, 512) for a CUDA Block size equal to 128. (Right) Performance (speedup over sequential execution) achieved by the *Block-Shared* implementation, *Flat*, *Full-Interleaved* (Full-Inter) and *Multicore* (Multi) using 16 cores. The test-case consisted of computing 256,000 medium-high neurons, using one of the two logic GPUs in one K80 NVIDIA GPU.

illustrates the performance achieved by the *Block-Shared* and the other approaches. Although using shared memory is better than the performance achieved by the *Flat* approach, it is much smaller than the *Full-Interleaved* counterpart. For this particular scenario (medium-high morphology), a very low number of systems saturate the capacity of the shared memory (48KB in NVIDIA k80). Also, the data reuse is low using one-thread per Hines system. These drawbacks do not allow to achieve a better performance when the shared memory is used. It is important to note that the usage of the shared memory affects to the exchanging of CUDA blocks to overlap computation with memory accesses.

Finally, we evaluate the impact on performance of the particularities of each of the morphologies (Table 4.1). The performance achieved by the *Flat* is not included as it was proven to be very inefficient. As shown in Fig. 4.8, both approaches, *Multicore* and *Full-Interleaved*, show a similar trend in performance independently of the neurons' morphology. In particular the peak speedup achieved on the different morphologies does not vary significantly (47×-55×).

After comparing the performance achieved by Multicore and GPU, now we focus on evaluating the efficiency of our GPU implementation. To do that, we make use of *NVPROF*[2]. We do not obtain results very different depending on the input (number and shape of neurons). In all cases, we obtain more than 99% of efficiency (sm_efficiency). It is also achieved a bandwidth (Global Load Throughput) close to 160GB/s, being the theoretical peak equal to 240 GB/s and the effective about the bandwidth achieved by our implementation. As most of the GPU applications, our implementation is memory bound and this is reflected by a low occupancy (about 24%).

[2]nvprof -m achieved_occupancy,sm_efficiency,gld_throughput,gst_throughput,gld_efficiency, gst_efficiency ./run

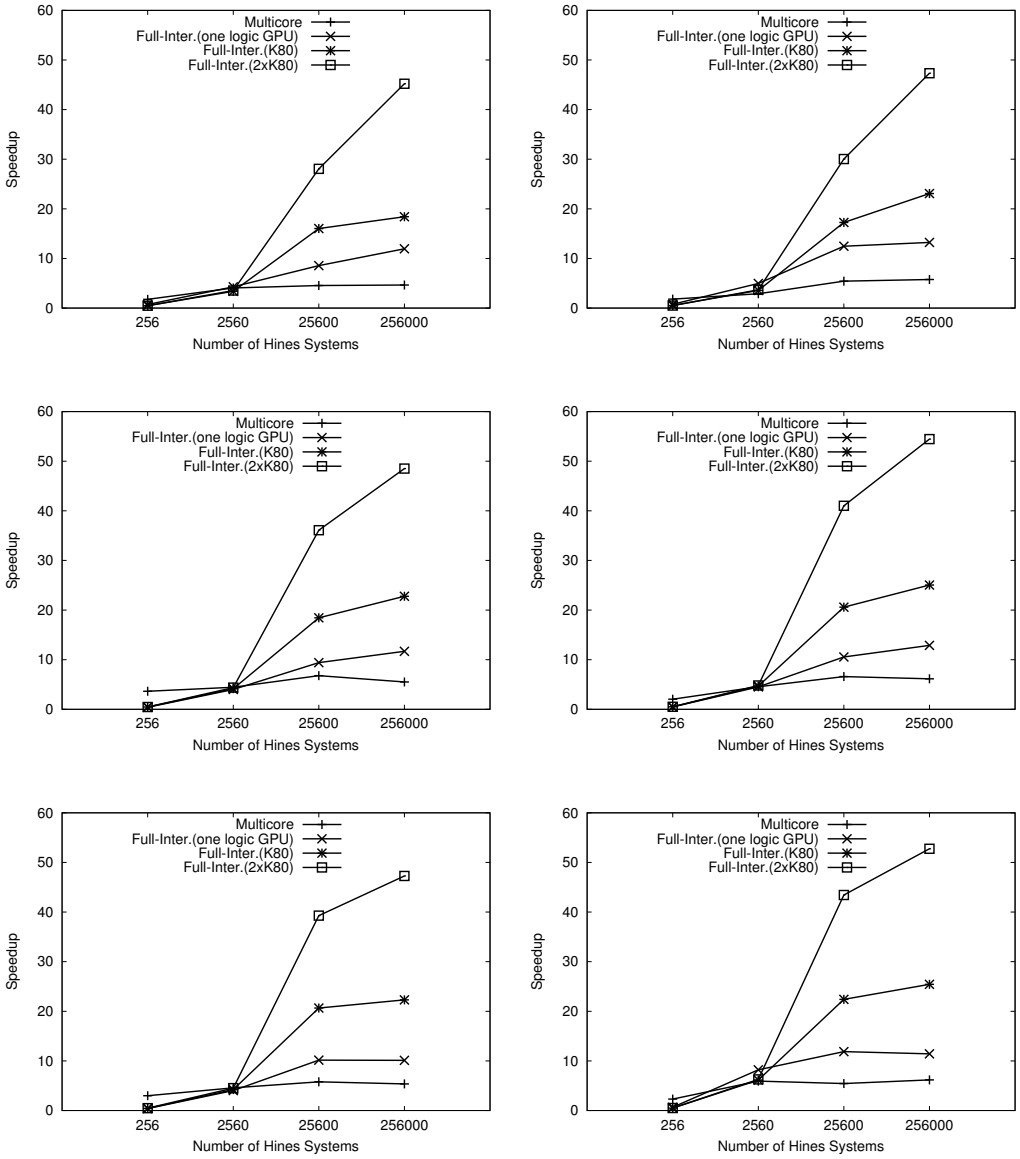

Figure 4.8: Performance (speedup over sequential execution) achieved for computing multiple (256, 2,560, 25,600, 256,000) neurons using different morphologies: *small-low* (top-left), *small-high* (top-right), *medium-low* (center-left), *medium-high* (center-right), *big-low* (bottom-left) and *big-high* (bottom-right).

4.5 cuHinesVBatch

As it was done in Section 4.3, we explored the impact of dealing with neurons of different size, but unlike in cuThomasVBatch, here we have to deal with an additional drawback, the differences found in the neurons' morphologies. In order to evaluate the impact of both parameters we just generated synthetic morphologies based on some neurons' properties and % of branching given as an input.

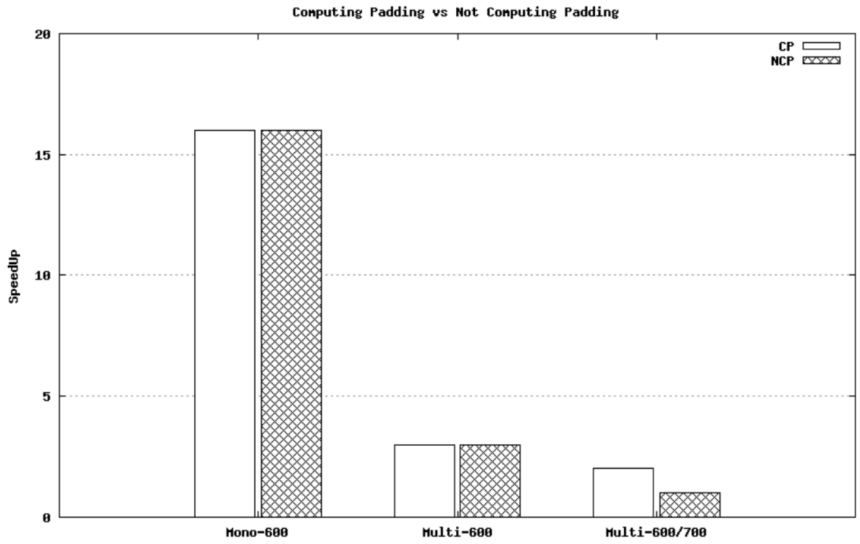

Figure 4.9: Execution time for the two variants, *No Computing Padding* and *Computing Padding*, of the *cuHinesVBatch*, for neurons with same morphology(mono), different morphology(multi) and different size and morphology(multi 600/700).

As we can see in Figure 4.9, the fall in performance for neurons of the same size (600) with different morphologies is critically high, leading us to explore another approaches and opening the way of the future work.

Chapter 5

Conclusions and Future Work

Regarding cuThomasBatch, our implementation is able to outperform the *cuSPARSE* implementation, even when a low number of systems is computed. This is because of a simpler management of CUDA threads, as we do not have to deal with synchronizations, atomic operations and the limitations regarding the size of shared memory and CUDA blocks. It is important to remark that the implementation of the *cuThomasBatch* code presented in this paper will be included in the next cuSPARSE release as *gtsvInterleavedBatch*, being this a great success.

Otherwise, the performance obtained by cuHinesBatch for more relevant inputs(neurons with different morphologies and different sizes) is still under our expectations, for this reasons as future work, we want to analyze different approaches in order to expose a higher parallelism with a lower number of neurons.

The integration of cuHinesBatch into the Arbor simulator is still under development and it is being carried out by ETH Zürich and the Swiss National Supercomputing Center (CSCS) as the main developers of Arbor. That is the main reason why we could not present any performance results regarding the simulation itself, despite being the main objectives of this work.

Chapter 6

Contributions

In this Section we are going to review all the contributions derived from the presented work.

- Pedro Valero-Lara, Ivan Martinez-Perez, Raul Sirvent, Xavier Martorell, Antonio J. Pena. NVIDIA GPUs scalability to solve multiple (batch) tridiagonal systems. Implementation of cuThomasBatch . 12th International Conference on Parallel Processing and Applied Mathematics (PPAM), September 10-13, 2017 At Lublin, Poland.

  The solving of tridiagonal systems is one of the most computationally expensive parts in many applications, so that multiple studies have explored the use of NVIDIA GPUs to accelerate such computation. However, these studies have mainly focused on using parallel algorithms to compute such systems, which can efficiently exploit the shared memory and are able to saturate the GPUs capacity with a low number of systems, presenting a poor scalability when dealing with a relatively high number of systems. The gtsvStridedBatch *routine in the* cuSPARSE *NVIDIA package is one of these examples, which is used as reference in this paper. We propose a new implementation (*cuThomasBatch*) based on the Thomas algorithm. Unlike other algorithms, the Thomas algorithm is sequential, and so a coarse-grained approach is implemented where one CUDA thread solves a complete tridiagonal system instead of one CUDA block as in* gtsvStridedBatch*. To achieve a good scalability using this approach is necessary to carry out a transformation in the way that the inputs are stored in memory to exploit coalescence (contiguous threads access to contiguous memory locations). Different variants regarding the transformation of the data are explored in detail. The results given in this study prove that the implementation carried out in this work is able to beat the reference code, being up to 5× (in double precision) and 6× (in single precision) faster using the latest NVIDIA GPU architecture, the Pascal P100.*

- Pedro Valero-Lara, Ivan Martinez-Perez, Antonio J. Pena, Xavier Martorell, Raul Sirvent, and Jesus Labarta. cuhinesbatch: Solving multiple hines systems on gpus human brain project* . In International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland, pages 566–575, 2017.

  The simulation of the behavior of the Human Brain is one of the most important challenges today in computing. The main problem consists of finding efficient ways to manipulate and compute the huge volume of data that this kind of simulations need, using the current technology. In this sense, this work is focused

on one of the main steps of such simulation, which consists of computing the Voltage on neurons' morphology. This is carried out using the Hines Algorithm. Although this algorithm is the optimum method in terms of number of operations, it is in need of non-trivial modifications to be efficiently parallelized on NVIDIA GPUs. We proposed several optimizations to accelerate this algorithm on GPU-based architectures, exploring the limitations of both, method and architecture, to be able to solve efficiently a high number of Hines systems (neurons). Each of the optimizations are deeply analyzed and described. To evaluate the impact of the optimizations on real inputs, we have used 6 different morphologies in terms of size and branches. Our studies have proven that the optimizations proposed in the present work can achieve a high performance on those computations with a high number of neurons, being our GPU implementations about 4x and 8x faster than the OpenMP multicore implementation (16 cores), using one and two K80 NVIDIA GPUs respectively. Also, it is important to highlight that these optimizations can continue scaling even when dealing with number of neurons.

- Pedro Valero-Lara, Ivan Martinez-Perez, Raul Sirvent, Xavier Martorell, Antonio J. Pena. cuThomasBatch: a new CUDA Routine to compute Multiple(Batch) Tridiagonal Systems on NVIDIA GPUs. CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE 2018;00:1-12.

  The solving of tridiagonal systems is one of the most computationally expensive parts in many applications, so that multiple studies have explored the use of NVIDIA GPUs to accelerate such computation. However, these studies have mainly focused on using parallel algorithms to compute such systems, which can efficiently exploit the shared memory and are able to saturate the GPUs capacity with a low number of systems, presenting a poor scalability when dealing with a relatively high number of systems. The gtsvStridedBatch *routine in the* cuSPARSE *NVIDIA package is one of these examples, which is used as reference in this paper. We propose a new implementation (*cuThomasBatch*) based on the Thomas algorithm. Unlike other algorithms, the Thomas algorithm is sequential, and so a coarse-grained approach is implemented where one CUDA thread solves a complete tridiagonal system instead of one CUDA block as in* gtsvStridedBatch*. To achieve a good scalability using this approach is necessary to carry out a transformation in the way that the inputs are stored in memory to exploit coalescence (contiguous threads access to contiguous memory locations). Different variants regarding the transformation of the data are explored in detail. The results given in this study prove that the implementation carried out in this work is able to beat the reference code, being up to 5× (in double precision) and 6× (in single precision) faster using the latest NVIDIA GPU architecture, the Pascal P100.*

- cuThomasBatch will be added in the next cuSPARSE release under the name gtsvInterleavedBatch.
- cuThomasBatch-cuThomasvBatch[1] repositories
- cuHinesBatch[2] repositories

[1] `https://pm.bsc.es/gitlab/run-math/cuThomasBatch-cuThomasVBatch`
[2] `https://pm.bsc.es/gitlab/imartin1/cuHinesBatch`

Acknowledgements

To complete this project just need to thank all those people who have made this project could end in fruition.

To Pedro, my director, for providing me this opportunity, give me the confidence and guidelines that have allowed much of our analysis.

To my friends Artem and Navarro, for help me when work was accumulated

My girlfriend Gala, for dealing with my frustration when things not came out and stay awake at night next to me while working.

And finally my parents, who have teach me the value of hard work and bring me the opportunities in order to follow my dreams.

Bibliography

[1] Roy Ben-Shalom, Gilad Liberman, and Alon Korngreen. Accelerating compartmental modeling on a graphical processing unit. *Frontiers in Neuroanatomy*, 7:4, 2013.

[2] Samuel Daniel Conte and Carl W. De Boor. *Elementary Numerical Analysis: An Algorithmic Approach.* McGraw-Hill Higher Education, 3rd edition, 1980. ISBN 0070124477.

[3] cuSPARSE. Nvidia-cuda toolkit documentation. *http://docs.nvidia.com/cuda/cusparse/.*

[4] Andrew Davidson, Yao Zhang, and John D. Owens. An auto-tuned method for solving large tridiagonal systems on the GPU. In *IEEE International Parallel and Distributed Processing Symposium*, May 2011.

[5] Andrew A. Davidson, Yao Zhang, and John D. Owens. An auto-tuned method for solving large tridiagonal systems on the GPU. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS, Anchorage, Alaska, USA*, pages 956–965, May 2011.

[6] Sandra Diaz-Pier, Mikaël Naveau, Markus Butz-Ostendorf, and Abigail Morrison. Automatic generation of connectivity for large-scale neuronal network models through structural plasticity. *Frontiers in Neuroanatomy*, 10:57, 2016. ISSN 1662-5129. doi: 10.3389/fnana.2016.00057. URL `http://journal.frontiersin.org/article/10.3389/fnana.2016.00057`.

[7] Michael Hines. Efficient computation of branched nerve equations. *International Journal of Bio-Medical Computing*, 15(1):69 – 76, 1984. ISSN 0020-7101. doi: http://dx.doi.org/10.1016/0020-7101(84)90008-4. URL `http://www.sciencedirect.com/science/article/pii/0020710184900084`.

[8] Hee-Seok Kim, Shengzhao Wu, Li wen Chang, and Wen mei W. Hwu. A scalable tridiagonal solver for GPUs. *2013 42nd International Conference on Parallel Processing*, 0:444–453, 2011. ISSN 0190-3918. doi: http://doi.ieeecomputersociety.org/10.1109/ICPP.2011.41.

[9] NVIDIA. The most advanced datacenter accelerator ever built featuring pascal gp100, the world's fastest gpu. In *White paper: NVIDIA Tesla P100*, pages 1–45, 2017.

[10] N. Sakharnykh. Efficient tridiagonal solvers for adi methods and fluid simulation. In *NVIDIA GPU Technology Conference*, September 2010.

[11] Harold S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *J. ACM*, 20(1):27–38, January 1973. ISSN 0004-5411. doi: 10.1145/321738.321741. URL `http://doi.acm.org/10.1145/321738.321741`.

[12] Pedro Valero-Lara. Multi-gpu acceleration of DARTEL (early detection of alzheimer). In *2014 IEEE International Conference on Cluster Computing, CLUSTER 2014, Madrid, Spain, September 22-26, 2014*, pages 346–354, 2014.

[13] Pedro Valero-Lara and Fernando L. Pelayo. Towards a more efficient use of gpus. In *International Conference on Computational Science and Its Applications, ICCSA 2011, Santander, Spain, June 20-23, 2011*, pages 3–9, 2011.

[14] Pedro Valero-Lara and Fernando L. Pelayo. Analysis in performance and new model for multiple kernels executions on many-core architectures. In *IEEE 12th International Conference on Cognitive Informatics and Cognitive Computing, ICCI*CC 2013, New York, NY, USA, July 16-18, 2013*, pages 189–194, 2013.

[15] Pedro Valero-Lara, Alfredo Pinelli, Julien Favier, and Manuel Prieto Matias. Block tridiagonal solvers on heterogeneous architectures. In *IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, ISPA '12, pages 609–616, 2012.

[16] Pedro Valero-Lara, Alfredo Pinelli, and Manuel Prieto-Matias. Fast finite difference poisson solvers on heterogeneous architectures. *Computer Physics Communications*, 185(4):1265 – 1272, 2014. ISSN 0010-4655. doi: 10.1016/j.cpc.2013.12.026.

[17] Pedro Valero-Lara, Alfredo Pinelli, and Manuel Prieto-Matías. Fast finite difference poisson solvers on heterogeneous architectures. *Computer Physics Communications*, 185(4):1265–1272, 2014.

[18] Pedro Valero-Lara, Poornima Nookala, Fernando L. Pelayo, Johan Jansson, Serapheim Dimitropoulos, and Ioan Raicu. Many-task computing on many-core architectures. *Scalable Computing: Practice and Experience*, 17(1):32–46, 2016.

[19] Pedro Valero-Lara, Ivan Martínez-Perez, Antonio J. Peña, Xavier Martorell, Raúl Sirvent, and Jesús Labarta. cuhinesbatch: Solving multiple hines systems on gpus human brain project*. In *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*, pages 566–575, 2017.

[20] Pedro Valero-Lara, Alfredo Pinelli, Julien Favier, and Manuel Prieto-Matías. Block tridiagonal solvers on heterogeneous architectures. In *10th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA, Leganes, Madrid, Spain*, pages 609–616, July 2012.

[21] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the GPU. *SIGPLAN Not.*, 45(5):127–136, January 2010. ISSN 0362-1340. doi: 10.1145/1837853.1693472. URL `http://doi.acm.org/10.1145/1837853.1693472`.

[22] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the GPU. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP, Bangalore, India*, pages 127–136, January 2010.